M.S. THESIS

Utilizing Genetic Algorithm to LambdaMART Forests to Predict Ranking Labels Accurately

랭킹 라벨을 정확히 예측하기 위한 유전 알고리즘의 LambdaMART 포레스트에 대한 적용

June 2017

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Jeong Wonil

M.S. THESIS

Utilizing Genetic Algorithm to LambdaMART Forests to Predict Ranking Labels Accurately

랭킹 라벨을 정확히 예측하기 위한 유전 알고리즘의 LambdaMART 포레스트에 대한 적용

June 2017

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Jeong Wonil

Utilizing Genetic Algorithm to LambdaMART Forests to Predict Ranking Labels Accurately

랭킹 라벨을 정확히 예측하기 위한 유전 알고리즘의 LambdaMART 포레스트에 대한 적용

지도교수 Srinivasa Rao Satti

이 논문을 공학석사학위논문으로 제출함

2017 년 6 월

서울대학교 대학원

컴퓨터 공학부

정원일

정원일의 석사학위논문을 인준함

2017 년 6 월

위 원 장	Bernhard Egger	(인)
부위원장	Srinivasa Rao Satti	(인)
위 원	허충길	(인)

Abstract

Utilizing Genetic Algorithm to LambdaMART Forests to Predict Ranking Labels Accurately

Jeong Wonil
School of Computer Science Engineering
Collage of Engineering
The Graduate School
Seoul National University

In this thesis, principles of genetic algorithm (GA) will be applied to forests of LambdaMART to get more accurate ranking results. Ranking problem is considered one kind of prediction function problems, and various solutions were proposed for the ranking problem. Applying machine learning techniques has improved ranking quality of algorithm. One of the techniques is ensemble of decision tree learning where each tree is trained one by one and these trees are used to predict the result with the given input values.

LambdaMART is a fusion of LambdaRank and MART (Multiple Additive Regression Trees), where gradients of scores are calculated by LambdaRank and multiple trees are generated and trained with predefined steps in MART. LambdaMART is also main contributor for the winner of "Yahoo! Learning to Rank Challenge (2010)" though the challenge reports that ranking solution performance has reached saturation point. However, LambdaMART might have problems about overfitting to training data, which means it could not predict outcome precisely on other unobserved data after being trained with data. In

addition, genetic algorithm can provide greater searching ability for solution space though the ability depends on designing core operations such as crossover, mutation, and so on.

Combining this search ability with LambdaMART could enhance solution's quality and reduce some chance of overfitting to training data. Each LambdaMART forest will become a chromosome in this scheme, and multiple forests will be operands of genetic operations. This scheme shows higher accuracy measure value than original LambdaMART and total training time per forest has also been saved.

Keywords: LambdaMART, Genetic Algorithm, Learning, Ranking, Regression tree

Student Number: 2015-21262

Contents

List of Figures

List of Tables

Chapter 1

Introduction

These days machine learning techniques have become more popular than ever, and machine learning technique can be applied to solve ranking problem. Machine learning means a scheme or a technique where predictions about unknown data are made by training its system with lots of known data. Machine learning includes neural networks, deep learning, and decision tree learning. Ranking is one of elemental constituents for web information retrieval system, and ranking requires system to retrieve most relevant documents and they should be ranked by their degree of relevance according to queries issued by user. To solve this ranking problem, learning algorithms have been proposed and showed their effectiveness to the problem.

Some of the learning algorithms applied an ensemble of multiple learning algorithms using decision tree. GBRT [5], and LambdaMART [2] used ensemble of decision trees for learning. The algorithm with decision tree trains decision tree where features of training samples become classifier for output space. When the purpose of a decision tree is to predict discrete or non-continuous values, the decision tree is called classification tree, and regression tree otherwise.

Several algorithms utilized multiple decision trees such as random forest,

bootstrap aggregating (bagging), bayesian model combination, and gradient boosting. These algorithms usually aim to obtain solution of higher quality than that of consisting learning algorithm alone. LambdaMART is one of those algorithms with gradient boosting. LambdaMART is proven to be the most important contributing algorithm for winner of "Yahoo! Learning to Rank Challenge (2010)" [4] where participant's algorithm will be trained with large training data and then evaluated with testing data, then the algorithm that results in the highest ranking measure value becomes the winner. The Yahoo! Learning to Rank Challenge reports that the winner utilized LambdaMART tree models trained with NDCG [9], however the top participants algorithm showed quite similar performance which implies that the ranking algorithms are close to saturation point.

Genetic algorithm [1] is a powerful tool to solve complex problems. Though it depends on design of operations, genetic algorithm can provide efficient searching ability for solution space. Combining this searching ability with LambdaMART is expected to improve performance. The main contribution of this thesis comes from the invention of the fusion algorithm of LambdaMART and GA, and the new algorithm shows overall better ranking measure values for larger training data when the number of trees of a forest is the same. To make better performance for the combination, various genetic operations and parameters for probability have been searched. In addition, the total time taken for training per forest has been saved compared to the original LambdaMART. The results and structure of this combined method will be explained in the following chapters.

Chapter 2 deals with background knowledge such as ranking problem, Classification and Regression Trees (CART) [13], and genetic algorithms. Chapter 3 introduces related works, and then structure of invented algorithm will be shown in Chapter 4. Chapter 5 elaborates experimental results and settings. Chapter 6 concludes this thesis and suggests future works.

Chapter 2

Background

2.1 Information Retrieval: Ranking

Information Retireval (IR) is to acquire information resources from a collection of data according to the requests of users, and includes searching for metadata, and ranking [14]. Ranking is one of the most fundamental operations in Information Retrieval (IR) since the ranking operation sorts the retrieved documents by their relevance to the need of user. Therefore, ranking operation is also a core function for widespread applications such as social networking systems, internet search, and recommender systems. Following subsections will represent the problem and measures for the problem.

2.1.1 Ranking Problem

In ranking problem, given a query q and a collection D of documents that match the query, the algorithm should sort the documents according to relevance grades or other criteria so that best and superior results are positioned at the front side of result list. Usually, the document set is modelled as vector space model in which feature values of a document become a vector since cal-

culating numeric scores on query-document pairs is main issue. For this reason, ranking problem can be thought of as function estimation problem or predictive learning problem. In predictive learning problem, a system should obtain approximation function $\hat{F}(x)$ of the ideal function $\bar{F}(x)$ where mapping of feature value vector $\boldsymbol{X}$ to relevance grade y should minimize expected value of a specified loss function $L(y, F(\boldsymbol{X}))$ after being trained with N training samples $(\boldsymbol{X}, y)_1^N = \{x_1, x_2, x_3, ..., x_n, y\}_1^N$.

$$\bar{F}(x) = \underset{F}{\operatorname{argmin}} E_{y,x}\, L(y, F(\boldsymbol{X})) \tag{2.1}$$

For ranking problem, approximation function should sort documents according to trained system and minimize cost function which can be squared error function such as $(y - F(x))^2$, and y-value is limited to relevance grade value. Also, alternative approach can be applied to this problem where a score function is defined on pairs of documents d_f, d_r and get gradients of this function for sorting. Publicly available ranking challenge usually assigns 5 levels of relevance grades $(Relevance) = \{0, 1, 2, 3, 4\}$ to documents which are results of a query, and a document is represented as feature ID - value pairs such as $\{\mathbf{1}, 0.5314\}$ in a document.

2.1.2 Ranking Measures

NDCG is the abbreviation for "Normalized Discounted Cumulative Gain". DCG (Discounted Cumulative Gain) is calculated as the following equation. Cumulative Gain at k-th rank position is defined as

$$CG_k = \sum_{i=0}^{k} rel_i \tag{2.2}$$

rel_i means relevance grade of the result at position i, and it is not related to ordering of documents. For DCG, the relevance grade value is reduced as the position of result is further from the first position. DCG at k is defined as the

following.

$$DCG_k = \sum_{i=1}^{k} \frac{rel_i}{\log_2(i+1)} \tag{2.3}$$

Alternative definition of DCG at k is

$$DCG_k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)} \tag{2.4}$$

For the second formulation the base of log does not change NDCG value whereas the first formulation can change NDCG. As DCG value is the highest when the position of documents are sorted by their relevance grades, DCG indicates accuracy of the document sorting for a given query. Since the length of result list can be different from query to query, DCG at k should be normalized across queries to compare search performance consistently among all queries. For this reason, the DCG values are normalized by using IDCG (Ideal Discounted Cumulative Gain) value which is the maximum DCG value with the query.

$$NDCG_k = \frac{DCG_k}{IDCG_k} \tag{2.5}$$

$$IDCG_k = \sum_{i=1}^{k_{rel}} \frac{2^{rel_i} - 1}{\log_2(i+1)} \tag{2.6}$$

k_{rel} is the position at k after ideally sorting relevant documents.

With this NDCG value, a ranking algorithm can compare its performance among all different queries and this comparison is significant for training since calculation of loss function requires the evaluation of current scores.

MAP (Mean Average Precision) is the mean value of average precision over all queries. Average precision is calculated as average of precisions on documents resulted from one query. Precision at k is defined as the precision until k-th position in the list. Thus, the formulation for AP (average precision), and MAP (mean average precision) are the following.

$$AP = \frac{\sum_{i=1}^{n}(Precision(i) \times rel(i))}{(D_{rel})} \tag{2.7}$$

where n is the number of retrieved documents, and $rel(i)$ is indication function that has 1 if the document is related or 0 otherwise. D_{rel} is the number of related documents.

$$MAP = \frac{\sum_{q=1}^{Q} AP(q)}{Q}, \tag{2.8}$$

where Q is the number of queries. Suppose that documents in the following table are positioned by an algorithm and marked each document as relevant or irrelevant with the help of real evaluation of the documents.

Table 2.1 Sorting of documents retrieved for query 1

Sorted Position	Document ID	Relevance
1	1001	true
2	1004	true
3	1006	false
4	1007	true
5	1005	false
6	1002	false
7	1003	true

Precision of all the documents in the table 2.1.2 are calculated by putting denominator as the total number of retrieved documents and numerator as the total number of relevant and retrieved documents before evaluation of average precision. All precision values are $P(1) = \frac{1}{1}$, $P(2) = \frac{2}{2}$, $P(3) = \frac{2}{3}$, $P(4) = \frac{3}{4}$, $P(5) = \frac{3}{5}$, $P(6) = \frac{3}{6}$, and $P(7) = \frac{4}{7}$. Average precision is the average of the precision values $\frac{\sum_{i=1}^{6} P(i)}{6} \approx 0.6623$. Likewise, the AP of the table 2.1.2 is 0.6857. Thus, MAP is $\frac{0.6623+0.6857}{2} = 0.6740$.

Table 2.2 Sorting of documents retrieved for query 2

Sorted Position	Document ID	Relevance
1	1005	true
2	1001	false
3	1002	true
4	1006	true
5	1004	false
6	1003	true
7	1007	false

2.2 Classification and Regression Trees

Decision tree is a tree where each internal node determines true or false of a predicate and leaf node shows the label of result. When decision tree is used for data mining or training it is called classification tree when variables can have discrete values, regression trees when variables can have continuous values [13]. In decision tree learning, multiple input (x_i, y) values denoted as $(\boldsymbol{X}, y) = \{x_1, x_2, x_3, ..., x_n, y\}$ are training samples and the objective is to expect unobserved y values as accurately as possible with the training samples. In classification tree, the y values can be one of the class $\boldsymbol{C} = \{C_1, C_2, ..., C_c\}$ while y values can have real value in regression tree. When training continues, a node can be split to divide solution space and this can be repeated recursively until the solution space of a node has the same value with the target variable or it meets end conditions.

2.3 Genetic Algorithm (GA)

Genetic Algorithm (GA) is a kind of evolutionary algorithm where a solution entity called chromosome is designed and multiple chromosomes interact with each other or with environment to have better solution quality. They compete with

each other by replacement or selection operations, generate new chromosome through crossover, and get changed by mutation. Some variant of GA adopts local optimization after classical genetic operations to make chromosomes closer to local optimums in the solution space. After going through several stages of operations, the structure of the best chromosome will be a solution of the algorithm and usually the result solution is very close to global optimal solution's performance, especially when solution space is very large and difficult to find solution.

2.3.1 Selection

Selection operator chooses multiple chromosomes from genetic pool where all chromosomes on the current generation are listed with some standard or rules. In most cases, two chromosomes are chosen so that those chromosomes are usually superior than other chromosomes. Several schemes have been introduced for this operator where chromosomes with higher quality have higher probability to be selected.

Roulette Wheel method [18] assigns probability proportional to the fitness or quality of chromosomes. Then, it selects one or two chromosomes randomly from genetic pool based on probabilities given to each chromosomes.

The figure 2.1 represents an example of roulette wheel selection with 4 chromosomes where each area is proportional to fitness of each chromosome.

Tournament Selection [17] holds tournament battle among chromosomes until top two winners are determined. It needs a parameter called selection pressure which is used to determine the probability of superior chromosome's winning in one competition. This operator can produce more diverse chromosomes.

2.3.2 Crossover

Crossover operator generates one or more new chromosomes by exchanging structure of the selected chromosomes. Usually, one child is generated from two parent chromosomes.

Single point crossover selects one dividing point at random on the structure of chromosomes, then one chromosome's structure before the splitting point and the other chromosome's structure after the point are combined together to generate new chromosome. In case of multiple point crossover [21], multiple points are selected and one chromosome's parts and the other chromosome's parts take turns for new chromosome according to the points.

Real value crossover scheme utilizes real valued components of chromosomes and generates new chromosome by choosing random value between parent's values. For example, if chromosome A has a real value 10, and the other chromosome B has a real value 25, the generated chromosome N can have a value

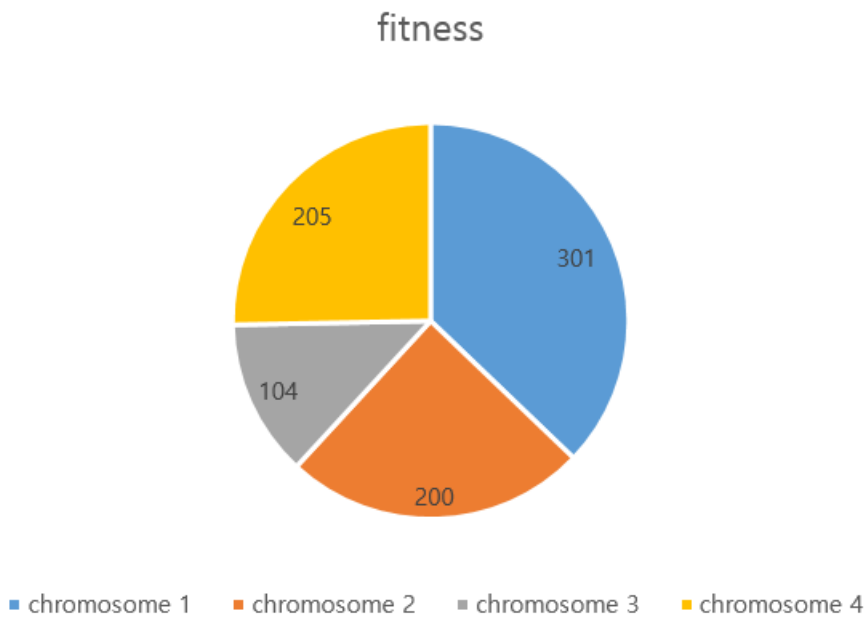

Figure 2.1 Roulette Wheel Selection probability chart with 4 chromosomes

between 10 and 25.

2.3.3 Mutation

Mutation operator [20] changes some or minor part of a chromosome or all chromosomes with a little probability. Usually the child chromosome is mutated after crossover operation. The purpose of this operator is to prevent the system from converging into local optimums. It also can help system to search solution space better.

One of the general ways is to change some part of a new chromosome. On the other hand, mutation operator will mutate all chromosomes in the current generation. For the latter case, the probability should be more smaller than the former.

2.3.4 Replacement

Replacement operator replaces one chromosome or multiple chromosomes from parent generation with the child chromosome made by crossover. This operator usually chooses inferior chromosome to be replaced with. One of the schemes frequently used is that worst chromosome in the genetic pool is substituted for the child chromosome. Another method replaces the worst chromosome among the parents and the child. Also, one can check which chromosome from the parents and the child is the worst and if the child is the worst chromosome replaces the worst chromosome on the genetic pool.

Chapter 3

Related Work

3.1 RankNet

RankNet [3, 11] can adopt any model only when the output of the model can be differentiated with the model parameters. In RankNet, the training data is divided by query. During training, RankNet assigns an input feature vector $\boldsymbol{x} \in \Re^d$ to $f(x)$ at some point. Documents are referred to as URL (Uniform Resource Locator), which indicates the address of a document with protocol name, hostname, and file name. When a query is given, a pair of URLs U_i and U_j with different labels are selected and given to the model. Then, the model with the pair computes the scores $s_i = f(x_i)$ and $s_j = f(x_j)$. After defining $U_i \triangleright U_j$ as an event that U_i should have higher rank than U_j, the two outputs of the model will be mapped to a trained probability that U_i should be ranked higher than U_j with a sigmoid function. Thus, the probability of $U_i \triangleright U_j$ is calculated as the following equation

$$P_{ij} \equiv P(U_i \triangleright U_j) \equiv \frac{1}{1 + e^{-\sigma(s_i - s_j)}} \tag{3.1}$$

where the parameter σ resolves the sigmoid shape. To assign penality to the deviation of the model output probabilities from the ideal probabilities, the

cross entropy cost function is devised, and the cost [6] is

$$C = -\bar{P}_{ij} log P_{ij} - (1 - \bar{P}_{ij}) log(1 - P_{ij}) \tag{3.2}$$

where $\bar{P}_{ij}$ is the known probability of U_i ranking higher than U_j. Let $S_{ij} \in \{0, \pm 1\}$ to be 1 if document i has been labeled to be more relevant than document j, -1 for the opposite case, and 0 if they have the same label [7]. Assuming desired ranking is known,

$$\bar{P}_{ij} = \frac{1}{2}(1 + S_{ij}) \tag{3.3}$$

Combining the above two equations makes

$$C = \frac{1}{2}(1 - S_{ij})\sigma(s_i - s_j) + log(1 + e^{-\sigma(s_i - s_j)}) \tag{3.4}$$

Since the cost becomes asymptotically linear (incorrect ranking) or zero (correct ranking),

$$\frac{\partial C}{\partial s_i} = \sigma\left(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{-\sigma(s_i - s_j)}}\right) = -\frac{\partial C}{\partial s_j} \tag{3.5}$$

To reduce the cost with stochastic gradient descent, this gradient will update the weights $w_k \in \Re$, which is the model parameters).

$$w_k \rightarrow w_k - \eta \frac{\partial C}{\partial w_k} = w_k - \eta\left(\frac{\partial C}{\partial s_i}\frac{\partial s_i}{\partial w_k} + \frac{\partial C}{\partial s_j}\frac{\partial s_j}{\partial w_k}\right) \tag{3.6}$$

where η is a positive learning rate and w_k is the weight of k-th decision tree. Thus,

$$\delta C = \sum_k \frac{\partial C}{\partial w_k}\delta w_k = \sum_k \frac{\partial C}{\partial w_k}\left(-\eta \frac{\partial C}{\partial w_k}\right) = -\eta \sum_k \left(\frac{\partial C}{\partial w_k}\right)^2 < 0 \tag{3.7}$$

To update the model, the gradient of the cost by the model parameters w_k must be specified, and before that, the gradient of the cost with respect to the model scores s_i is needed. The requirement to compute $\frac{\partial C}{\partial w_k}$ can be bypassed with gradient descent formulation of boosted trees by modelling $\frac{\partial C}{\partial s_i}$.

3.2 LambdaRank

One problem of RankNet is that optimization for the number of pairwise errors may not match well with some other IR measures. According to [3], LambdaRank is based on the fact that "the idea of writing down the desired gradients directly allows us to bypass the difficulties caused by the sorting of documents in most IR measures". Also, as [3] pointed out, "this does not indicate that the calculated LambdaRank gradients are not gradients of a cost". The work of LambdaRank found that only the gradients are needed to train a model, not the costs themselves. Let us define

$$\lambda_{ij} \equiv \frac{\partial C(s_{ij}s_j)}{\partial s_i} = \sigma\left(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{-\sigma(s_i - s_j)}}\right) \tag{3.8}$$

The change step λ for a specific URL U_1 is contributed from all other URLs for the same query that has different labels.

In practice, modifying (3.8) by multiplication of the size of the change in IR measure ($|\nabla_{IR}|$) produced by exchanging of the rank positions of U_1 and U_2 gives great results. For this reason, utility function $\bar{C}$ is invented such that

$$\lambda_{ij} \equiv \frac{\partial \bar{C}(s_{ij}s_j)}{\partial s_i} = \frac{-\sigma}{1 + e^{\sigma(s_i - s_j)}}|\nabla_{IR}| \tag{3.9}$$

In this case $\bar{C}$ should be maximized, equation (3.6) is replaced by

$$w_k \to w_k + \eta\frac{\partial \bar{C}}{\partial w_k} \tag{3.10}$$

which results in

$$\delta\bar{C} = \frac{\partial \bar{C}}{\partial w_k}\delta w_k = \eta\left(\frac{\partial \bar{C}}{\partial w_k}\right)^2 > 0 \tag{3.11}$$

As a result, even if IR measures are either flat or discontinuous on every point, LambdaRank can ignore this problem by computing the gradients after the URLs have been sorted by their scores [8]. Also, calculation of δs_i and δs_j shows that every pair generates an equal and opposite λ, and for a given URL, all the lambdas are incremented.

$$\delta s_i = \frac{\partial s_i}{\partial w_k}\delta w_k = \frac{\partial s_i}{\partial w_k}\left(-\eta\lambda\frac{\partial s_i}{\partial w_k}\right) \tag{3.12}$$

$$\delta s_j = \frac{\partial s_j}{\partial w_k}\delta w_k = \frac{\partial s_j}{\partial w_k}\left(\eta\lambda\frac{\partial s_j}{\partial w_k}\right) \tag{3.13}$$

3.3 MART (Multiple Additive Regression Tree)

MART is considered one of boosted tree models where the output is a linear combination of the outputs of a set of regression trees. We can think of a root node and two leaf nodes where all the data are residing on the root node and are waiting to be partitioned.

In a regression tree, the tree examines all samples and finds the threshold t such that all samples with $x_{ij} \leq t$ are categorized into the left child , and those otherwise fall to the right child, and should meet the condition that the sum

$$S_j \equiv \sum_{i\in Le}(y_i - \mu_{Le})^2 + \sum_{i\in Ri}(y_i - \mu_{Ri})^2 \tag{3.14}$$

is the lowest value possible. Le and Ri are the index sets of samples which belong to the left or right respectively, and μ_{Le} and μ_{Ri} are the mean of the label values on the set of samples that fall to the left and the right respectively. After finding the threshold for minimum S_j, the split with the threshold is attached to the root node. The two leaf nodes calculate γ_1, γ_2 , each of which is the average of the y-value of each side sample. In general, this whole process is repeated to form a tree with L leaves with $L-1$ iterations since one node exists before the first iteration.

MART is built with these regression trees where cost function is the least squares, and its output function $F(x)$ can be written as

$$F_N(x) = \sum_{i=1}^{N}\alpha_i f_i(x) \tag{3.15}$$

where $f_i(x) \in R$ is modelled by one regression tree and the $\alpha_i \in R$ is the weight value assigned to the i-th regression tree, and both $f_i(x)$ and α_i are trained by samples. The outputs of the tree are affected by a fixed value which is trained and related with each leaf node, $\gamma_{kn}, k = 1, ..., L, n = 1, ..., N$, where L means

the number of leaves and N the number of trees. User can choose parameters such as L, N and η. In MART, when k trees are trained, the next tree is trained with the gradient descent to decrease the value of loss function. The next tree is designed as the m derivatives of the cost with respect to the current model score which is evaluated at each training point $(\frac{\partial C}{\partial F_n}(x_i), i = 1, ..., m)$. Therefore,

$$\delta C \approx \frac{\partial C(F_n)}{\partial F_n} \delta F \tag{3.16}$$

As $\delta F = -\eta \frac{\partial C}{\partial F_n}$, $\delta C < 0$. Thus, each tree is designed to be the gradient of the cost function with respect to the model score. The new tree is added to the ensemble of trees with a step size $\eta\, \gamma_{kn}$, and γ_{kn} can be calculated either exactly or estimated by Newton's approximation.

3.4 LambdaMART

LambdaMART is a combined version of LambdaRank and Gradient boosted regression trees. An ensemble of LambdaMART algorithms showed the highest value of ranking measure on "Yahoo! Learning To Rank Challenge (2010)". LambdaRank calculates gradients of information retrieval function such as NDCG, which are not smooth functions, into smooth functions, and it makes the problems triggered by the sort in most IR measures negligible. LambdaRank is actually based on RankNet where some IR measures could not match well unless the optimization for the number of pairwise errors becomes the desired cost. LambdaMART continues its iterations until the number of trees becomes N, and it finds optimal dividing point and L leaf tree where a Newton step is assigned to be used for learning. The next iteration is affected by the previous iteration's step value and learning rate. The overall structure of LambdaMART is summarized below.

Algorithm 1 Algorithm: LambdaMART

Set number of trees N, number of training samples m, number of leaves per tree L, learning rate η

for $i = 0$ to m **do**
    $F_0(x_i) = \text{BaseModel}(x_i)$ // If BaseModel is empty, set $F_0(x_i) = 0$
end for
for $k = 1$ to N **do**
    for $i = 0$ to m **do**
        $y_i = \lambda_i$
        $w_i = \frac{\partial y_i}{\partial F_{k-1}(x_i)}$
    end for
    $\{R_{lk}\}_{l=1}^{L}$ // Create L leaf tree on $\{x_i, y_i\}_{i=1}^{m}$
    $\gamma = \frac{\sum_{x_i \in R_{lk}} y_i}{\sum_{x_i \in R_{lk}} w_I}$ // Assign leaf values based on Newton step.
    $F_k(x_i) = F_{k-1}(x_i) + \eta \sum_l \gamma_{lk} I(x_i \in R_{lk})$ // Take step with η.
end for

Chapter 4

LambdaMART with GA

4.1 Overview

In this scheme, multiple LambdaMART forests become genetic pool where each LambdaMART trainer is a chromosome. Parameters such as maximum tree depth, maximum number of leaves, the number of tree, and learning rate are also included in the chromosome. Initial genetic pool size is 10, where 9 forests get variated parameters of learning rate, maximum leaf number, and maximum tree depth. Initial values of these parameters will be variated by normal random distribution having mean value as the original value of the parameter and the standard deviation value as $\frac{1}{10}$ the original value. The remaining one constant forest has original parameter with learning rate $\eta = 0.01$, max leaf number $= 10$, max tree depth $= 4$, and tree number $T = 100$.

Training samples are divided by half, and the first half part of samples is processed alternating training and evaluating phase. The other half part is used for only training of forests, after the first half part is finished. The first half part is again partitioned into smaller parts, which are called stages. The stage is divided into two parts : training part and evaluating part. The system

parameter called training rate determines the ratio of training part over the stage and ranges from 0 to 1. Evaluating part obtains the remaining size of the stage. In this implementation, training rate is 0.75, which means 75% of the part is used for training and 25% for evaluation. In the training phase, all chromosomes including new chromosome from the previous stage will be trained with training samples from the stage. All forests are trained in parallel before the training of the new chromosome. After all forests are trained with the training part, genetic operators work on the genetic pool using evaluating part for calculating fitnesses. This algorithm changed the position of replacement operator to the first and only works if the previous stage has generated a new chromosome. Since the new chromosome generated should be evaluated for its training ability, the new chromosome also should experience training phase of a stage. For this reason, replacement operator is positioned before selection operator of the new stage. The figure 4.1 explains how original training data are divided and a stage is made.

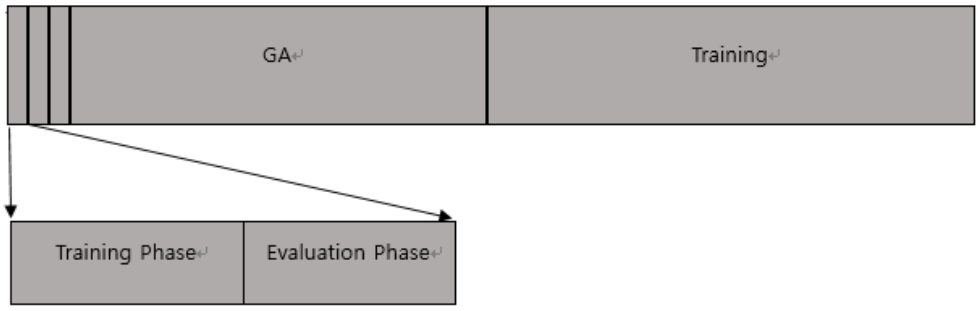

Figure 4.1 Overview of LambdaMART with GA

4.2 Genetic Operations

Genetic operations work on the evaluation phase of a stage. At first, evaluation of all forests including the child chromosome from the previous stage is executed to update average NDCG value calculated among all previous stages

so far. All stages except the first stage execute replacement operator following selection operator. Selection operator chooses 2 chromosomes, and makes a new chromosome with crossover operator. Mutation operator affects all chromosomes with minor probability, and it changes some part of step process by using random distribution. Evaluated samples from the stage are used for fitness calculation before all genetic operations, and the calculation results are used for the operations.

4.2.1 Selection

Roulette Wheel Selection method is adopted for the selection operator. Since NDCG ranges from 0 to 1 and even 0.01 improvement is significant as it approaches 1, using NDCG value directly for roulette probability is not effective. Roulette probability is scaled to be $2^{(NDCG/0.1)}$ to expand differences on NDCG value. The reason of scaling NDCG value difference is that the significance of the change increases as the NDCG value is close to saturation point or ideal NDCG value. Evaluation of chromosome should also consider this implicit difference, and it is also important for other genetic operations.

4.2.2 Crossover

The parameters of forests are used for crossover, and a new forest is generated by choosing a random value between parameter values of parent forests. Uniform random distribution is utilized for the selection of random value of the new forest. For example, when parent 1 has a learning rate 0.0101 and parent 2 has 0.0135, the child forest will select a real value between those parent values $0.0101 \leq$ (learning rate of child) ≤ 0.0135. The child forest's all parameters such as learning rate, maximum depth, and maximum leaf number are generated by the same way. After selection of chromosome parameters, the child should inherit the ensemble of trees from one of the parents so that the child can continue training on the next stage. In this implementation, referring to the

average NDCG value evaluated at the beginning of genetic operations, the new forest inherits the training trees of the parent which has worse average NDCG measure value, than the other. In this way, Overfitting into local maximum can be prevented.

4.2.3 Mutation

Mutation operator has two kinds of mutation which are reverse and random mutation. The probability of reverse and random mutation is 0.10 and 0.02 respectively. Reverse mutation changes direction of step size calculation with probability 0.10 by replacing subtraction with addition and addition with subtraction. The randomized step size calculation proceeds as the following equation.

$$\begin{aligned} response[jj] \mathrel{+}= \lambda &\rightarrow response[jj] \mathrel{-}= \lambda \\ response[kk] \mathrel{-}= \lambda &\rightarrow response[kk] \mathrel{+}= \lambda \end{aligned} \tag{4.1}$$

The respone values in this equation is the step value before scaling with learning rate. After multiplying this value with learning rate, the result is applied to the decision tree. On the other hand, random mutation makes a part of gradient calculation random with probability 0.05 by using normal random distribution with original value as mean value and a third of the original value as standard deviation. The randomized calculation is formulated as the following.

$$\rho = \frac{1.0}{(1.0 + e^{\text{normalRandom}(f[jj]-f[kk], \frac{f[jj]-f[kk]}{3})})} \tag{4.2}$$

The original part of this equation is

$$\rho = \frac{1.0}{(1.0 + e^{(f[jj]-f[kk])})} \tag{4.3}$$

where $fx[jj]$, $fx[kk]$ are functions for jjth document and kkth document respectively. The ρ value is multiplied by the difference of scores to become response value which is mentioned in the equation (4.1).

4.2.4 Replacement

The new chromosome from previous stage is evaluated first to update fitness value, check whether fitness value is higher than parent 1 and parent 2. The chromosome with the lowest fitness will be removed among parents and child, except for the child chromosome. When the child chromosome has the worst fitness value, it is compared with the chromosome owning the worst fitness value in the genetic pool, and the worst chromosome is substituted for the child chromosome if the child has a superior fitness value. For example, suppose parent 1 has fitness value 3.5 , parent 2 has 1.5, the worst chromosome has 0.7, and child has 1.4. Since the fitness value of child is worse than those of parents and the worst chromosome has lower fitness value than the child, the worst chromosome is replaced with the child chromosome.

Chapter 5

Experimental Results

5.1 System Settings and Datasets

Datasets used for experiment are"Yahoo! Learning to Rank Challenge", MQ2007 and MQ2008 in LETOR4.0 format. These datasets are publicly available and are extracted from real documents with the purpose of evaluating ranking ability of algorithms. Experiment is executed on the Linux server with Intel i7-7700K CPU (8-core, 4.20GHz), 32GB DDR4 RAM, and Linux kernel version 4.10.9.

5.2 Implementation

The invented combined algorithm is implemented by C++, and used C++ library for the original LambdaMART. Input and output functions are the same for the two libraries. Training dataset becomes input and the trained structure of the algorithm is saved to JSON file. After that, trained structure is loaded from the JSON file and use the loaded structure to get prediction values for the testing dataset which is also an input data.

The implementation of LambdaMART with GA is based on the C++ library of original LambdaMART. The process of training for all forests is invented and some low level functions are edited to execute mutation and to be convenient about the management of forests. Training of the multiple forestes is designed to be executed in parallel. OpenMP commands have been inserted to the part of code where looping for training of multiple forests executed. The version used for the implementation is 4.0. Through this parallel implementation, training time can decrease significantly.

5.3 Results

For convenience, LambdaMART is abbreviated as **LM** and LambdaMART with Genetic Algorithm as **LGM**. The following tables denote data size, NDCG, and training time of LambdaMART and LambdaMART with GA including **LM** with $T = 200$. Training time of **LGM** is divided by the number of forests to compare training time taken per forest. Split version of original dataset is used due to long time of training for original large data, except for MQ2007 and MQ2008. Training data and test data of the "Yahoo! Learning to Rank Challenge" are divided into 12000 lines and 4000 lines respectively.

Table 5.1 NDCG at 8 Result table including $T = 200$ results of original LambdaMART

NDCG at 8	LambdaMART(T=100)	LambdaMART with GA	LambdaMART(T=200)	Data Size(MByte)
Yahoo split1	0.6994	0.7060	0.7209	33.6
Yahoo split2	0.6325	0.6463	0.6741	30.8
Yahoo split3	0.5716	0.5814	0.5881	30.5
MSLR MQ2007 fold1	0.5150	0.5131	0.5234	25.8
MSLR MQ2007 fold2	0.4947	0.4994	0,4982	25.7
MSLR MQ2008 fold1	0.7139	0.6504	0.7046	5.9
MSLR MQ2008 fold2	0.6546	0.6497	0.6533	5.8

Table 5.2 Time taken per forest for NDCG at 8 including $T = 200$ results of original LambdaMART

NDCG at 8 time(sec)	LambdaMART(T=100)	LambdaMART with GA (per forest)	LambdaMART(T=200)
Yahoo split1	5713	2665	12608
Yahoo split2	3210	8041	6553
Yahoo split3	8209	2136	7053
MSLR MQ2007 fold1	11627	1689	22795
MSLR MQ2007 fold2	10822	5555	26107
MSLR MQ2008 fold1	756	575	1527
MSLR MQ2008 fold2	726	573	1523

Table 5.3 NDCG at 5 Result table including $T = 200$ results of original LambdaMART

NDCG at 5	LambdaMART(T=100)	LambdaMART with GA	LambdaMART(T=200)	Data Size(MByte)
Yahoo split1	0.6654	0.6684	0.7209	33.6
Yahoo split2	0.6175	0.5905	0.6395	30.5
Yahoo split3	0.5371	0.5886	0.5881	30.5
MSLR MQ2007 fold1	0.5012	0.4897	0.5074	25.8
MSLR MQ2007 fold2	0.4664	0.4908	0.4770	25.7
MSLR MQ2008 fold1	0.6671	0.6036	0.6752	5.9
MSLR MQ2008 fold2	0.6107	0.5887	0.6156	5.8

Table 5.4 Time taken per forest for NDCG at 5 including $T = 200$ results of original LambdaMART

NDCG at 8 time(sec)	LambdaMART(T=100)	LambdaMART with GA (per forest)	LambdaMART(T=200)
Yahoo split1	5552	2154	9782
Yahoo split2	3630	2066	6006
Yahoo split3	3694	2784	9594
MSLR MQ2007 fold1	6148	3561	6301
MSLR MQ2007 fold2	5803	4482	6012
MSLR MQ2008 fold1	352	337	708
MSLR MQ2008 fold2	341	298	773

From the tables 5.1-5.4, **LGM** usually shows ranking performance between **LM** with $T = 100$ and **LM** with $T = 200$ on Yahoo data. Since the number of trees influences ranking performance greatly, **LM** with $T = 200$ mostly performs the best. MQ2008 has the lowest data size among all datasets, and that is why the results on the MQ2008 datasets shows exceptional phenomena. Since **LGM** utilizes 10 LambdaMART forests, the total training time of **LGM** has been divided by 10 for fair comparison.

The figures 5.1-5.4 indicate the difference of the time taken for training per forest. More ranking measures such as NDCG at 5 with $T = 50$, NDCG at 8 with $T = 50$, NDCG at 5 with $T = 100$, and NDCG at 8 with $T = 100$ are included in the figures.

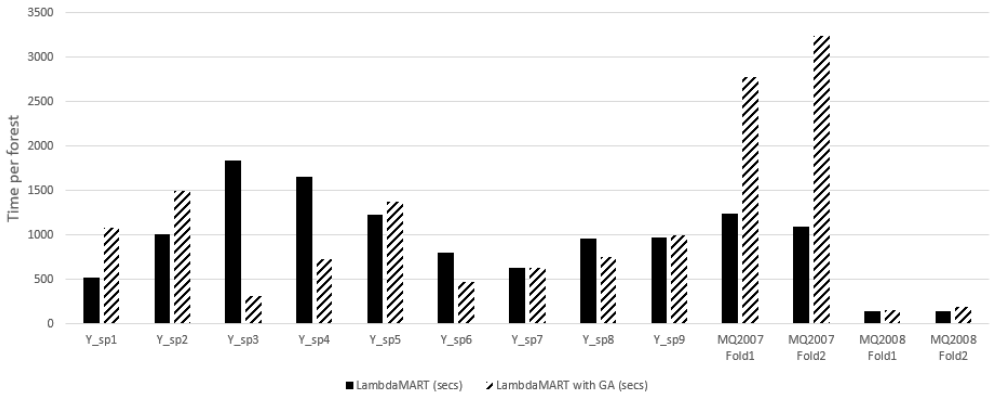

Figure 5.1 Time per forest for NDCG at 5 with $T = 50$

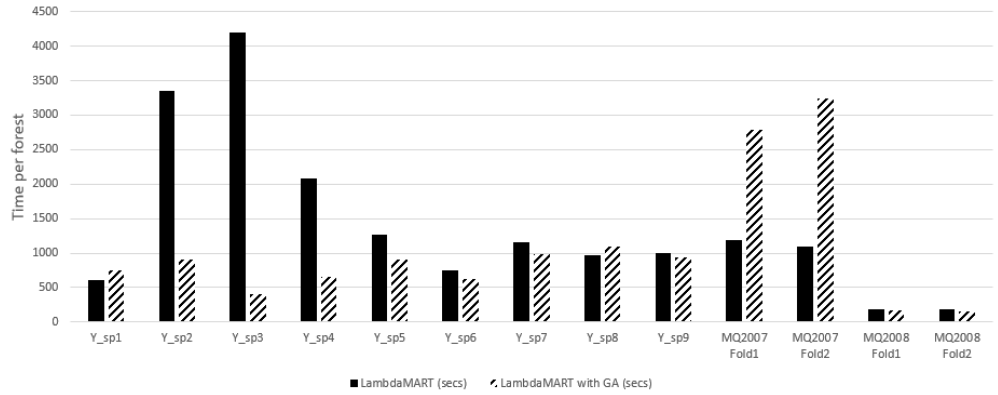

Figure 5.2 Time per forest for NDCG at 8 with $T = 50$

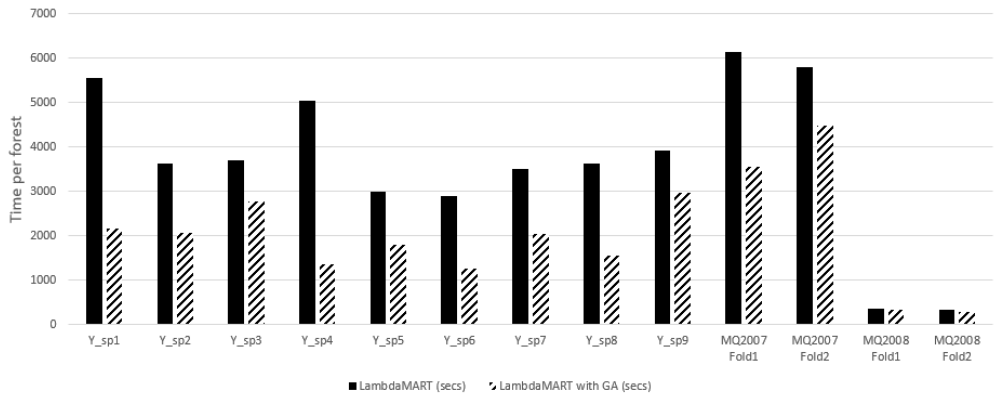

Figure 5.3 Time per forest for NDCG at 5 with $T = 100$

The figures 5.1-5.4 prove that **LGM** is usually better for the training time per forest. As shown later in the table 5.5, NDCG at 5 with $T = 50$ only shows worse time than **LM** and better time on other measures. The number of trees affects considerably on the time per forest because of implicit overhead in the **LGM** and it is easier to save time when one forest takes longer training time. In addition, the maximum position in the ranking measures such as NDCG at **5** have effects on the time, since the evaluation system should examine more sorted documents for larger required maximum position.

The difference of NDCG values are listed in the following figures. In this

case, the difference value is expanded by using power of 2 since the significance of difference should be emphasized.

The left bar of a column in the figures 5.5-5.8 is the training data size, and the right bar is the value of $2^{(\frac{NDCG_{diff}}{0.1})}$ multiplied by training data size. If the right bar of a column is higher than the left bar, it indicates **LGM** has been improved over **LM** considering the training data size. The difference of the performance depends on the feature of each dataset, and usually the smaller the training data, the less the ranking measures of **LGM**. In total, **LGM** has

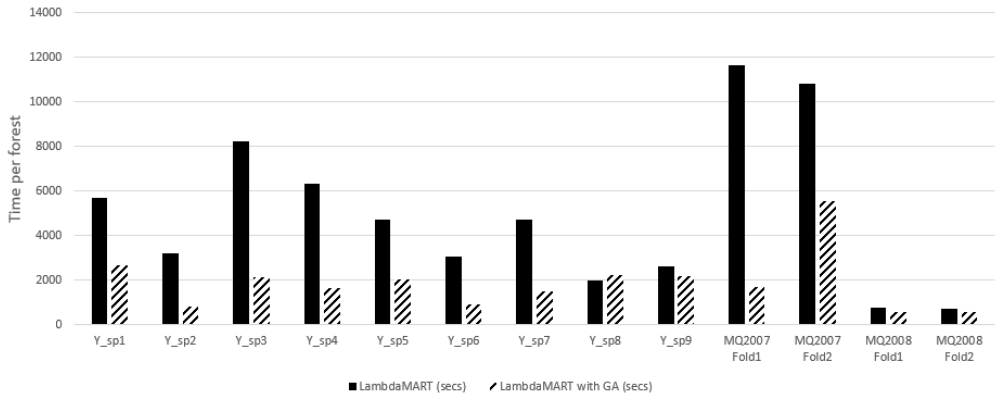

Figure 5.4 Time per forest for NDCG at 8 with $T = 100$

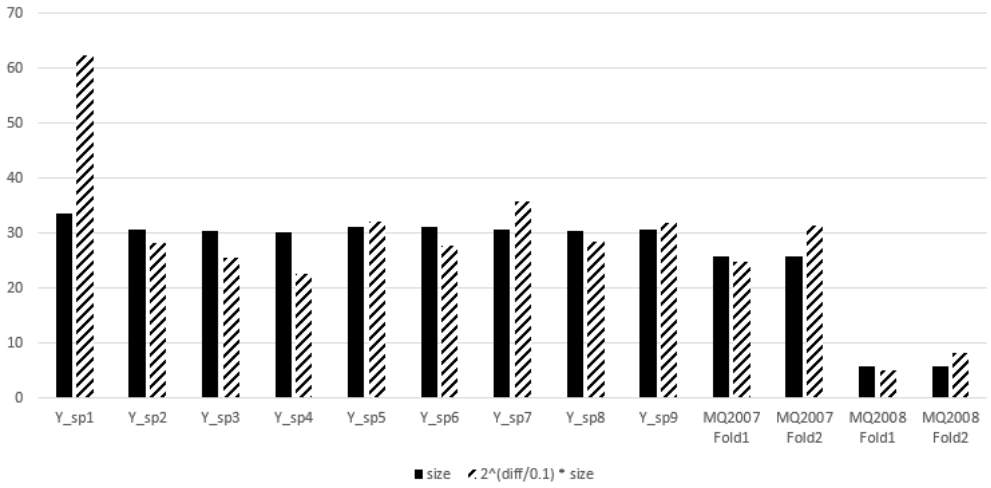

Figure 5.5 NDCG at 5 difference with $T = 50$ multiplied by data size

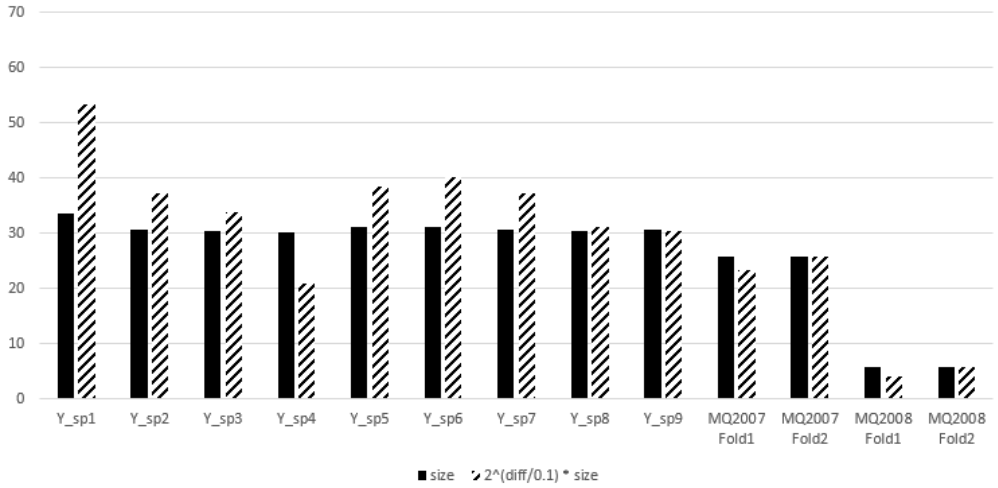

Figure 5.6 NDCG at 8 difference with $T = 50$ multiplied by data size

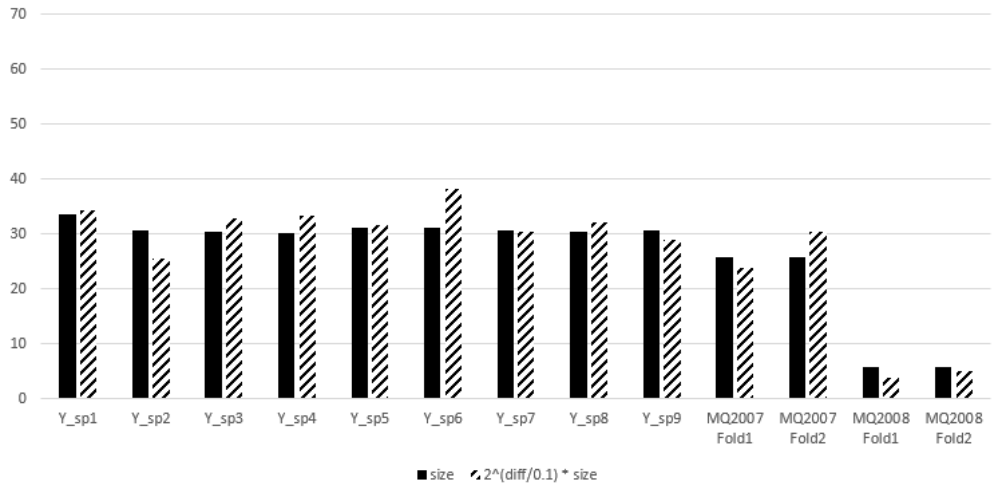

Figure 5.7 NDCG at 5 difference with $T = 100$ multiplied by data size

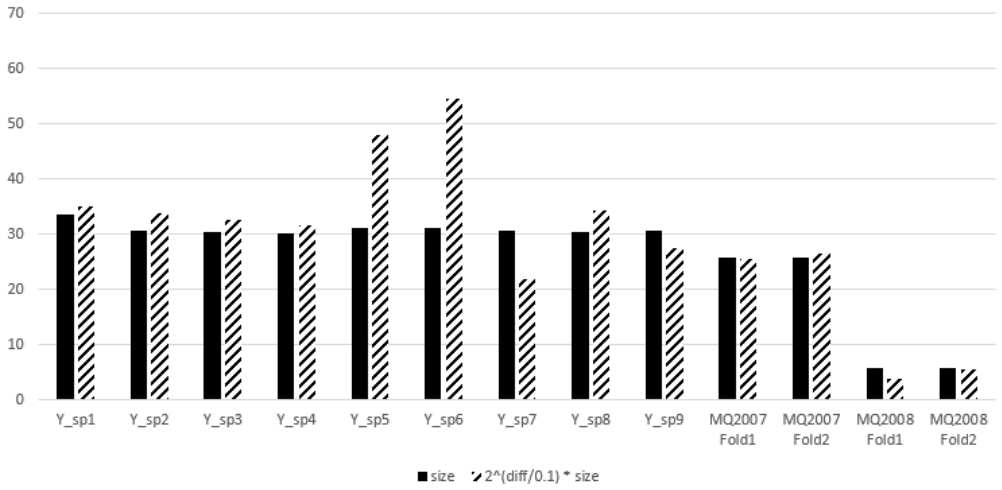

Figure 5.8 NDCG at 8 difference with $T = 100$ multiplied by data size

better ranking values than **LM** and **LGM** should improve its ranking quality on smaller datasets.

Average of each evaluation measures are summarized in the following table 5.5. The average value of $2^{(\frac{NDCG_{diff}}{0.1})} \times size$ is evaluated by multiplying each $2^{(\frac{NDCG_{diff}}{0.1})}$ value with training data size and then dividing it by the total sum of the size. $NDCG_{diff}$ is the difference of NDCG value calculated by subtracting NDCG of original LambdaMART from that of LambdaMART with GA.

Table 5.5 Average value of NDCG differences adjusted to see changes apparently and fairly

	$2^{(\frac{NDCG_{diff}}{0.1})}$	$2^{(\frac{NDCG_{diff}}{0.1})} \times size$	Time saved per forest (sec)
NDCG at 5 ($T = 50$)	0.06846	0.06365	-1960.6
NDCG at 5 ($T = 100$)	-0.00887	0.02428	20817.7
NDCG at 8 ($T = 50$)	0.07266	0.11511	4431.8
NDCG at 8 ($T = 100$)	0.07136	0.11286	40012

The table 5.5 reveals that **LGM** performs better than **LM** considering the training data size, which is denoted by the second column of this table. Time saved per forest also shows that **LGM** can save time compared to serial instances of forests.

LambdaMART with more trees shows worse performance than LambdaMART with less trees in the MQ2008 datasets, which is unusual case for ranking solution. One of the reasons is that MQ2008 dataset is suited for less fitting on the training set and more trees indicate increase of familiarity with training data. Also, MQ2008 datasets have much less size than others. LambdaMART with GA shows better results than LambdaMART with 100 trees and worse than LambdaMART with 200 trees on Yahoo split1 and Yahoo split3. As the size of dataset grows, the benefit of GA on LambdaMART becomes more apparent. Thus, applying genetic algorithm on LambdaMART will be helpful for training with large data, especially when a system unit can have limited number of trees.

Chapter 6

Conclusion

In this thesis, the principles of Genetic Algorithm have been applied to LambdaMART, which is combination of LambdaRank and MART. Genetic Algorithm is expected to increase searching ability of algorithm and increase performance results of LambdaMART which is the learning algorithm with multiple regression trees to solve ranking problem. Observing results of LambdaMART with GA, performance is enhanced for large training data whereas it did not show improvements on small training data. Considering saturation of ranking performance denoted in the report of "Yahoo! Learning to Rank Challenge", LambdaMART with GA achieved meaningful improvement for ranking problem. LambdaMART with GA can be useful for a system consisting of multiple computing units with limited number of trees since the results of each unit with a LambdaMART algorithm can be processed in GA and expect better results for large data context. The results indicate that ranking with large data will benefit from GA. For future study, improving ranking performance of the proposed method on smaller data should be done and training with other ranking measures should be tested.

Bibliography

[1] L.D. Davis, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

[2] Q. Wu, C.J.C. Burges, K. Svore and J. Gao, "Adapting Boosting for Information Retrieval Measures," *Journal of Information Regrieval*, 2007.

[3] C. Burges, "From RankNet to LambdaRank to LambdaMART: An Overview", Microsoft Research Technical Report MSR-TR-2010-82

[4] O. Chapelle, Y. Chang and T-Y Liu, "The Yahoo! Learning to Rank Challenge", http://learningtorankchallenge.yahoo.com, 2010.

[5] J.H. Friedman, "Greedy function approximation: A gradient boosting machine," IMS Reitz Lecture, Stanford, Technical Report, 1999.

[6] C.J. Burges,"Ranking as learning structured outputs, " in *Proceedings of the Neural Information Processing Systems workshop on Learning to Rank*, Whistler, BC, Canada, December 2005. pp. 7-11.

[7] C. Burges, R. Ragno, and Q. Le, "Learning to rank with non-smooth cost function, " in *Proceedings of the 19th International Conference on Neural Information Processing Systems*, 2006. pp. 193-200.

[8] P. Donmez, K.M. Svoer, and C.J.C. Burges, "On the local optimality of LambdaRank, " in *Proceedings of the 32nd international ACM SIGIR con-*

ference on Research and development in information retrieval, Boston, MA, USA, July 2009. pp.460-467

[9] K. Jarvelin and J. Kekalainen, "IR evaluation methods for retrieving highly relevant documents," in *Special Interest Group on Information Retrieval (SIGIR)*, New York, 2000. pp. 41-48.

[10] C.J.C. Burges, R. Ragno and Q.V. Le, "Learning to Rank with Non-Smooth Cost Functions," *Proceedings of the 2006 Conference on Advances in Neural Information Processing Systems 19*, London, England, 2006. pp. 193-200.

[11] C. Burges and others, "Learning to rank using gradient descent, " in *Proceedings of the 22nd international conference on Machine learning*, Bonn, Germany, August 2005. pp. 39-96.

[12] E.B. Baum and F. Wilczek, "Supervised Learning of Probability Distributions by Neural Networks, " in *Proceedings of the 1987 International Conference on Neural Information Processing Systems*, MIT Press Cambridge, MA, USA, 1988. pp. 52-61.

[13] L. Breiman and others, "Classification and Regression Trees," CRC Press, 1984.

[14] C.D. Manning, .P Raghavan and H. Schütze, "Introduction to information retrieval," June 12, 1992.

[15] J.E. Baker, "Adaptive selection methods for genetic algorithms, " Proceedings of the International Conference on Genetic Algorithms and Their Applications, 1985.

[16] M. Mitchell, J.P. Crutchfield, and R Das, "Evolving cellular automata with genetic algorithms: A review of recent work, ", in *Proceedings of the First*

International Conference on Evoluationary Computation and Its Applications(EvCA'96), Russian Academy of Sciences, Moscow, Russia, 1996.

[17] K. Deb and D.E. Goldberg, "A comparative analysis of selection schemes used in genetic algorithms, " in *Foundations of genetic algorithms*, 1991. pp. 69-93.

[18] J.E. Baker, "Reducing bias and inefficiency in the selection algorithm, ", in *Proceedings of the second international conference*, 1987. pp. 14-21.

[19] T. Blickle and L. Thiele, "A Comparison of Selection Schemes used in Genetic Algorithms, " Swiss Federal Institute of Technology(ETH), Computer Engineering and Communication Networks Lab(TIK),Zurich, Switzerland, TIK Report Nr. 11, December 1995

[20] H. Mühlenbein and D. Schlierkamp-Voosen, "Predictive models for the breeder genetic algorithm i. continuous parameter optimization, " in *Evolutionary computation* vol. 1, no.1, pp. 25-49, pp. 25-49, March 1993

[21] L. Booker, "Improving search in genetic algorithms, " in *Genetic algorithms and simulated annealing*, 1987. pp. 61-73.

요약

위 논문에서는, 더 정확한 랭킹 결과를 얻기위해 유전 알고리즘의 원리를 LambdaMART 포레스트에 적용하였다. 머신 러닝의 기술을 적용하는 것은 알고리즘의 랭킹 성능을 향상시켰으며, 그 예로 LambdaMART, 기울기 부스트 회귀 트리 (Gradiend Boosted Regression Trees : GBRT) 등이 있다. 머신 러닝 방법중에서 결정 트리 러닝 모음에서는 각각의 트리가 숙련이 되고 그 트리들이 인풋 값이 주어졌을 때 결과를 예측하는 데에 쓰인다. LambdaMART는 LambdaRank와 MART가 융합한 형태인 데, 문서의 점수 기울기는 LambdaRank로 계산하고, 다중 가합 회귀 트리(Multiple Additive Regression Trees : MART)에서 여러 개의 트리가 생성되고 미리 정해둔 스탭만큼씩 훈련된다.“Yahoo! Learning to Rank Challenge (2010)” 에서 비록 그 챌린지에서 랭킹 솔루션의 성능이 포화상태에 도달했다고 보고하였지만, LambdaMART는 승자의 알고리즘에 중요한 기여를 하였다. 유전 알고리즘은 비록 교차, 돌연변이 등과 같은 핵심 연산의 디자인에 영향을 받지만 솔루션 공간에 대한 상당한 탐색능력을 제공할 수 있다. 이러한 탐색 능력과 LambdaMART를 결합한다면 솔루션의 성능을 향상하고 훈련 데이터에 대한 오버피팅의 가능성을 줄일 수 있다. 이 논문의 제안하는 방식에서는 하나의 LambdaMART 포레스트가 하나의 염색체가 되어, 여러 포레스트들이 유전 연산의 대상이 될 것이다. 이 방식은 하나의 LambdaMART보다 높은 정확도 측정 수치를 보였으며, 한 포레스트 당 걸린 트레이닝 시간도 절약하였다.

주요어: LambdaMART, 유전 알고리즘, 랭킹, 학습, 회귀 트리

학번: 2015-21262